Big O and Recursion

THURSDAY, APRIL 24

Section materials curated by Jonathan Coronado, Yasmine Alonso, and Sean Szumlanski, drawing upon materials from previous quarters.

This week's section exercises provide practice with Big-O and begin our exploration of recursion with some interesting problems!

Remember that every week we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a **.cpp** file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

📦 Starter project

1. Big O, basics

Topics: Big-O, code analysis

What is the Big O runtime of the following functions, in terms of the variable N?

Code Snippet A

```
int sum = 0;
for (int i = 1; i <= N + 2; i++) {
    sum++;
}
for (int j = 1; j <= N * 2; j++) {
    sum++;
}
```

Code Snippet B

```
int sum = 0;
for (int i = 1; i <= N - 5; i++) {
    for (int j = 1; j <= N - 5; j += 2) {
        sum++;
    }
}
```

Code Snippet C

```
int sum = 0;
for (int i = 1; i <= 10000000; i++) {
    sum ++;
}
```

Code Snippet D

```
int sum = 0;
for (int i = 0; i < 1000000; i++) {
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
}
```

Code Snippet E

```
int sum = 0;
for (int i = 1; i <= N; i *= 2) {
    sum ++;
}
```

Solution

```
Code Snippet A has a runtime complexity of O(N): The first for loop
runs in O(N),
the second also runs in O(N). This makes O(N) + O(N) = O(2N) = O(N),
because we
throw away constants

Code Snippet B has a runtime complexity of O(N^2): The code in the
innermost for
loop is just a sum so that runs in constant time, O(1). The j for loop
iterates
a total of (N - 5)/2 times which is equal to O(N) since we ignore
constants and
scalars. The outermost for loop also runs in O(N). In total we get
O(1) * O(N) * O(N) = O(N^2)

Code Snippet C has a runtime complexity of O(1): This loop doesn't
depend on N
at all, and loops up to a constant 10000000. Therefore runtime is O(1)

Code Snippet D has a runtime complexity of O(1): This is also constant
time. The
outermost for loop loops up until 1000000, which is constant. The 3
inner loops
all loop until i, which will be always less than 1000000. Therefore
total
runtime is O(1)

Code Snippet E has a runtime complexity of O(logN): This runs in
logarithmic
time, O(logN). You can see this by counting the number of times it
takes i to get to
n. i starts at 1, then jumps to 2, then 4, ..., n. It takes logN steps
to get
to N. This is similar to logarithmic runtime analysis in class that
starts from
n and drops down to 1.
```

2. Big O, Recursive Functions - Fibonacci

Topics: Big-O, code analysis, recursion

What is the Big O runtime of the following function, in terms of the variable n?

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Solution

```
The time complexity for fibonacci is exponential, O(2^n). This is
tricky!
We can speculate that it's exponential because 1 recursive call makes 2
others which in turn make 4 others and so on. In general this is how
you
can analyze Big O of recursive functions. First, find what the runtime
is in 1 recursive call. Next, count how many recursive calls we make in
terms of N.

In a single recursive call to fibonacci, we just make other calls to
Fibonacci.
So outside of the recursive calls, we do constant time amount of work.
Next,
let's analyze how many calls we make in terms of n. To do that, it
helps to
try a few examples. Fibonacci(3) calls Fibonacci(2) and Fibonacci(1).
Fibonacci(2) then calls Fibonacci(1) and Fibonacci(0). In total, for N
= 3,
we make 4 recursive calls. Let's try another example for Fibonacci(4).
Fibonacci(4) calls Fibonacci(3) and Fibonacci(2). From the above, we
know
that Fibonacci(3) results in 4 recursive calls. Fibonacci(2) then calls
Fibonacci(1) and Fibonacci(0). In total, we make 8 recursive calls.

Do you see some kind of pattern here? From our simple examples, it
appears
for each n, we make approximately 2^(n - 1) recursive calls (It's
approximate
because this formula doesn't fit for all n). We can write 2^(n - 1) as
2^n / 2. Therefore we can write the Big O as O(1)[for amount of work in
a
single recursive call] * O(2^n / 2)[the total number of recursive calls
we make].
Remember that in Big O, we discard constant multipliers, so the Big O
for fibonacci simplifies to O(2^n).
```

3. Win some, lose sum (**sum.cpp**)

Topics: recursive backtracking

Write a recursive function named **canMakeSum** that takes a reference to a **Vector<int>** and an

`int` target value and returns true if it is possible to have some selection of values from the `Vector` that sum up to the target value. In particular, you should be implementing a function with the following declaration

```
bool canMakeSum(Vector<int>& values, int target)
```

For example, let's say that we executed the following code

```
Vector<int> nums = {1,1,2,3,5};
canMakeSum(nums, 9)
```

We should expect that the call to **canMakeSum** should return true. Given the values specified in **nums**, we can select 1, 3, and 5 such that **5 + 3 + 1 = 9**.

However, let's say that we executed the following code instead

```
Vector<int> nums = {1,4,5,6};
canMakeSum(nums, 8);
```

We should expect that the call to **canMakeSum** in this case should return false, since there is no possible combination of values from the vector that sum up to the target value of 8.

Solution

```
// SOLUTION 1
bool canMakeSumHelper(Vector<int>& v, int target, int sumSoFar) {
        if (v.isEmpty()) {
                return sumSoFar == target;
        }
        /* Here we choose the last element in the vector.
         * We could have chosen any element, but the last
         * is the easiest and fastest method.
         */
        int choice = v[v.size() - 1];
        v.remove(v.size() - 1);

        bool with = canMakeSumHelper(v, target, sumSoFar + choice);
        bool without = canMakeSumHelper(v, target, sumSoFar);

        // And then we unchoose, by adding this back!
        v.add(choice);

        return with || without;
}

bool canMakeSum(Vector<int>& v, int target) {
        return canMakeSumHelper(v, target, 0);
}

// SOLUTION 2
/*
 * This solution is similar to the one above, except it uses
 * an additional index parameter in a vector to make the choices,
 * instead of removing from the vector like solution 1 did.
 */
bool canMakeSumHelper(Vector<int>& v, int target, int sumSoFar, int
index) {
        if (index >= v.size()) {
                return sumSoFar == target;
        }

        // This is our choice now. Remember we can choose any element
        // in the vector, so we choose the element at 'index'
        int choice = v[index];

        bool with = canMakeSumHelper(v, target, sumSoFar + choice,
index + 1);
        bool without = canMakeSumHelper(v, target, sumSoFar, index +
1);

        // We don't have to add back, because we never removed!
        return with || without;
}

bool canMakeSum(Vector<int>& v, int target) {
        return canMakeSumHelper(v, target, 0, 0);
}

// SOLUTION 3
bool canMakeSumHelper(Vector<int>& v, int target, int sumSoFar) {
    // We are only looking for one sum. If we already have a sum that
matches
    // the target, there's no need to explore all the subsets (i.e, get
to the
    // case when the underlying vector is empty). We can stop early!
```

```
// Please note that this doesn't apply to all subset generation problems. The
// only reason we can do this is because we are looking for just one sum!
if (sumSoFar == target) {
    return true;
}

if (v.isEmpty()) {
    // I'm leaving this to preserve the code from Solution 1.
    // But returning false here yields the correct results too!
    // Do you see why ?
    return sumSoFar == target;
}

/* Here we choose the last element in the vector.
 * We could have chosen any element, but the last
 * is the easiest and fastest method.
 */
int choice = v[v.size() - 1];
v.remove(v.size() - 1);

bool with = canMakeSumHelper(v, target, sumSoFar + choice);
// We are only looking for one sum. If we can make the sum in the
// 'with' case, there's no need to try exploring the 'without' case
if (with) {
    // restore the underlying vector before returning. This is not
    // needed for correctness, but the vector is passed by reference
    // to this recursive function. So it will be very bad to remove
    // some elements from it permanently.
    v.add(choice);
    return true;
}

bool without = canMakeSumHelper(v, target, sumSoFar);
// And then we unchoose, by adding this back!
v.add(choice);

// I'm leaving this to preserve the code from Solution 1.
// But returning 'without' here yields the correct results too!
// Do you see why ?
return with || without;
}
```

Follow up: can you make the above a bit cleaner with short-circuting? Next, can you attempt making solution 2 efficient too?

---

4. Shrink Words (**`shrink.cpp`**)

Topics: recursive backtracking

Write a recursive function named **`shrinkWord`** that takes a **`string`** word and a **`Lexicon`** of words, and shrinks the word to the smallest possible word that can be found in the Lexicon. You can think of a the lexicon as a set of all words in the English dictionary. Checkout the documentation for Lexicon to learn more.

```
string shrinkWord(string input, Lexicon& lex)
```

The function is best described by this example below:

Given a string **"starter"**, we can shrink it to **"a"** through these: **starter** -> **starer** (by removing the second **t**) -> **stare** (by removing the second **r**) -> **tare** (by removing **s**) -> **are** (by removing **t**) -> **ae** (by removing **r**) -> **a** (by removing **e**). Hence, we'll return **"a"**. Note that all the intermediate words are english words.

As another example, given string **"baker"**, we can shrink it to **"bake"** through these: **baker** -> **bake** (remove **r**). We can't shrink any further because if we remove a any another letter, we can't find the resulting word in the lexicon. Hence, we'll return **"bake"**.

Finally, for **"fishpond"**, we can't make a single transformation. So we'll return **"fishpond"**, unchanged.

Solution

To shrink a word, we'd to remove a letter from the word. The question is, which letter should we remove? This is what our backtracking solution explores!

```
string shrinkWord(string input, Lexicon& lex) {
        // We can't further shrink an empty word.
        // Also, if current word we have now is not
        // an English word(i.e. it isn't contained in
        // the lexicon), that's also invalid(from the problem
        // description. )
    if (input.empty() || !lex.contains(input)) {
        return "";
    }

    string shortestWord = input;
    for (size_t i = 0; i < input.length(); i ++) {
                // Remove the letter at index i and recurse!
        string subword = input.substr(0, i) + input.substr(i + 1);
        string result = shrinkWord(subword, lex);

                // Compare the words we've generated so far to
                // our shortestWord. If our new word is smaller,
                // we have to change our shortestWord!
                // We need a special check for the empty string
                // because our base case returns "" for invalid inputs.
        if (!result.empty() && result.length() < shortestWord.length())
{
            shortestWord = result;
        }
    }
    return shortestWord;
}
```


Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-22